Bigloo: Mixing Posix Threads and Fair Threads

Réunion ParTout 1 an

CNAM, Paris III, 25 janvier 2010

Avant : 2 APIs exclusives

- 2 implémentations natives
  - Sur threads natifs (Posix, Java, .NET)
  - Gestion des locks par appels natifs
- backend utilisé (pthread, fthread ou nothread)
  - Variable globale
  - Constructeur unique (`make-thread` *thunk*)

Idée : utiliser l'API pthread

- Ré-implémenter la partie native des fthread au-dessus des pthread
- Code 100% en Bigloo Scheme
- Réutilisation, et maintenance facilitée
- Suppression du backend « fthread »
  - `(instantiate::`***`type`*** `(body` *`thunk`*`))`
  - Primitives `(current-thread)`
    `(thread-yield!)`
    `...`

Implémentation : fair thread

- ```
  (abstract-class thread
      (name (default (gensym 'thread)))
      ...)
  ```
- ```
  (class fthread::thread
      (scheduler (default #f))
      (body::procedure read-only)
      (%builtin (default #f)); native thread
      ...)
  ```
- ```
  (class %pthread::pthread
      (fthread (default #f)); current-thread
      (mutex (default (make-mutex)))
      (condvar (default (make-condvar)))
      ...)
  ```
  ```

Scheduler

- Ne sont pas mappés sur des pthreads
- Invoqué par **`await!`** ou **`yield!`**
- Exécuté par le fthread courant

  →Election d'un autre fthread à exécuter

  →Fin de l'instant courant (si aucun élu)
-
```
(class scheduler::fthread
   (body (lambda () (schedule self)))
   (%builtin (instantiate::%pthread
        (body (lambda () #unspecified)))))
```

Exemple d'appel au scheduler

```
(thread-yield!)
 (%thread-cooperate [th (current-thread)])
  (%scheduler-switch-to-next-th th scdl)
   (let ((nt (%scheduler-next-th scdl)))
    (%pthread-switch %th %nt)
     (with-access::%nth (cv mutex)
      (condvar-signal! cv mutex)))
    (%pthread-wait %th)
     (with-access::%th (condv mutex)
      (condvar-wait! condv mutex))))))
```

L'opération « unlink »

- `(make-asynchronous-signal proc)`
  → Procécude à un argument, le signal
- Exécution d'une procédure « détachée »
- Opération « longue » / durée d'un instant
- Création d'un pthread à l'instant suivant
- Invoque **`emit!`** à la terminaison
- ```
  (thread-await!
     (make-asynchronous-signal
        (lambda (s) (read))
  ```
  ```

(foo (|| (bar) (gee)))

- ```
  (let*
    ((s1 (make-async-signal (λ(s) (bar))))
     (s2 (make-async-signal (λ(s) (gee))))
     (sigs (list s1 s2)))

    (multiple-value-bind (val1 sig1)
      (thread-await*! sigs)
      (foo val1)
      (multiple-value-bind (val2 sig2)
        (thread-await*! sigs)
        (foo val2))))
  ```
  ```

Perspectives

- Orc + Hop : sujet de mon stage de Master 2
- Compilation vers automates ? (FunLoft, DSL)